AIAA-99-4342

AN OBJECT-ORIENTED DESIGN FOR RAPID MODIFICATION OF FILTERS

Michael M. Madden*

Unisys Corporation
20 Research Drive
Hampton, VA 23666

Abstract

Using object-oriented languages, developers can package data and functions into classes and relate them by the services they provide. Classes that provide the same service but use different implementations are placed in inheritance relationships. The base class declares the interface for the service, and the derived classes define the implementation. Thus, client code can be written to the unified interface, but the system behavior depends on the actual class that is constructed. This feature is called abstraction [1]. Abstraction removes decisions about system configuration from the code that models operation to the code that handles construction. The operational code is easier to read and maintain. The operational code is also more efficient because it does not redundantly evaluate configuration logic every frame. Control system development and research can benefit from the application of abstraction. In simulations used for control system research and development, transfer function definitions frequently change. The change can occur in the coefficients, the order of the transfer function, or the digital algorithm. The simulation may retain several transfer function configurations for comparative analysis or to accommodate multiple research projects. Applying abstraction to the design of transfer functions keeps the logic that manages different configurations out of the operational code.

Introduction

This paper examines the advantages that object-oriented design brings to the rapid modification of transfer functions, called filters within this paper. Rapid modification is the ease by which the definition of a filter can be found and changed *correctly*†. A filter's definition includes its order, its coefficients, and the digital algorithm that implements the transfer function (e.g. a Tustin z-transform). The paper also explores a related topic, the simplicity of maintaining multiple filter configurations. Maintenance of multiple configurations includes the ease with which a new configuration can be created and the ease of identifying and correctly modifying existing configurations.

Rapid modification and multiple configurations are important in simulation projects that involve control system‡ development or research. In these projects, filter definitions frequently change; and multiple filter configurations are retained for comparison. To highlight the advantages that object-oriented design brings to this environment, the object-oriented design is compared to a procedural design. The procedural design originates from the Langley Standard Real-time Simulation (LaSRS), a simulation framework written in FORTRAN. The LaSRS design is not unique; similar implementations have appeared in simulation code re-

* Senior Member, AIAA

† Correctness is an important criterion. Between designs that aid change, a design that inhibits unintentional defects is superior to one that facilitates them.

‡ This paper defines a control system as a collection of models that simulate from the pilot controls to the control effectors; e.g., stick dynamics, control laws, and actuator models.

ceived from third parties. The object-oriented design comes from the successor to LaSRS, LaSRS++, written using C++.

The Procedural Design

LaSRS implements each pair of transfer function order (e.g. first order, second order) and digital algorithm as a function. For example, the function signature for a first order filter implemented with a Tustin z-transform is:

```
FUNCTION FOTUST(U,A,B,C,D,SCR)
```

The arguments are defined as follows: U is the input to the filter. A, B, C, and D are the Laplace-domain coefficients. SCR is an array of at least six elements in length; the array holds the equivalent z-domain coefficients and the past values of the input and output. The subroutine also has a hidden dependence on the variable T (elapsed time) and H (time step); these are accessed globally through a common block. The function returns the output of the filter calculation. The procedural design has two characteristics that complicate filter modification and maintenance of multiple configurations. First, the filter function is separate from its implementation data, i.e. the scratch array. Second, the function tightly binds a filter's definition to its execution.

The developer must create and manage a "scratch" array for each filter. The size of the scratch array differs by transfer function order (since order dictates the number of internal states that must be stored). If a new release of the control system changes the order of the transfer function, the developer must remember to change the size of the scratch array in addition to the subroutine call. Otherwise, a defect will be introduced; the program will read and write past an array boundary. Alternatively, the developer could make all "scratch" arrays large enough to accommodate the highest order filter supported by the framework. However, this increases the memory requirements for the simulation unnecessarily. The scratch array being distinct from the filter function also opens the possibility that a scratch array will accidentally be used more than once, also introducing a defect; later filters will incorrectly use the internal states and temporary calculations of earlier filters. The potential defects associated with scratch arrays are very difficult to uncover.

Multiple configurations further complicate the management of scratch arrays. The developer can create a new scratch array for each filter of each configuration; this significantly increases the number of scratch arrays that are maintained. Or, the developer can create one scratch array for all configurations of a given filter; the developer must still remember to correctly size the array each time a new configuration is added.

The filter function defines and executes the filter. Thus, the filter definition is ingrained in the control system operational code. If a change is made to either the transfer function or the digital algorithm of a filter, the change appears in the operational code. Since filter executions are dispersed throughout the operational code, the definitions are also isolated; the developer cannot collect filter definitions in a central location for easy inspection and modification. In large control systems, the developer will traverse thousands of lines of code to find the execution point of the filters to be changed.

The isolation of filter definitions presents a problem for supporting multiple configurations. Multiple configurations are typically selected using conditional statements (e.g. IF-THEN). The developer can place a conditional statement at the execution point of each filter in the configuration. This effectively litters the operational code with conditional statements, reducing performance and readability. To identify the configuration, all conditional statements must be located and analyzed. Alternatively, the developer can copy and tailor the operational code for each configuration. Then, one conditional statement selectively executes the operational code tailored for a given configuration. This option significantly increases code size and creates a maintenance problem. The developer must replicate changes to the operational code across all copies. In both cases, the simulation evaluates the configuration each frame of operation even though the configuration will not change during operation. The extra computation is inefficient and is better placed in the initialization code. But, the procedural design prevents it.

Representing the transfer function coefficients with variables can transplant part of the filter configuration to the initialization code. The coefficient definitions are relocated from the operational code to the initialization

code. Any change to a filter coefficient is now made in the initialization code; the operational code remains unmodified. Configurations that differ only by the coefficients are evaluated once in the initialization code; the operational code no longer performs this evaluation. However, changing the transfer function order or the digital algorithm still requires a change to the operational code. In fact, a different function is called. At best, the procedural design allows only part of the filter definition to be configurable during initialization.

The following examples summarize the steps involved in modifying filters and creating a new configuration; they represent worst case condition: order, coefficients, and digital algorithm are changed. The order of the transfer function is increased in the example.

Changing an existing filter

1) Search the operational code for the filter.
2) Change the name of the filter function to the function appropriate for the new order and digital algorithm.
3) Create new variable names for the additional coefficients.
4) Note the name of the scratch array and new coefficient variables.
5) Search the declaration code for the scratch array and expand its size for the new filter function.
6) Search for the coefficient declarations and declare the new variables.
7) Modify initialization code for the new coefficient variables.

Creating a new configuration

1) For each filter in the configuration[§]:
   a) Search the operational code for the filter.
   b) Add a conditional statement for selecting the configuration.
   c) Add the function call for the new filter.
   d) To minimize data size, reuse as many variables from the other configurations and perform steps 3-7 in the "Changing an existing filter."

   The new scratch array must be large enough to support all configurations.

These steps show that the procedural design requires changes to multiple units of code (e.g. declaration, initialization, and operation) when a filter is changed. Depending on the development process, each of these units may require review, testing, and verification. Modified code units and their dependents must also be re-compiled; as more code units are changed, the set of files that are re-compiled grows. If multiple programmers develop the operational code, placing filter definitions in the operational code also impacts configuration management. The operational code is very volatile during development. The procedural design adds filter definitions and multiple configurations as potential changes to the operational code. Thus, the procedural design can significantly increase the potential changes that must be reconciled when the work of multiple developers is combined. The more changes that must be reconciled in a unit of code, the larger the chance that a defect will be introduced during the merge process. All these factors indicate that maintenance of the procedural design can be costly.

The Object-Oriented Design

The object-oriented design eliminates the disadvantages of the procedural design by exhibiting the opposite characteristics. In other words, the implementation data is bound to the implementation; and the filter definition is separate from its execution. This is accomplished using abstraction mechanisms of encapsulation and polymorphism [1]. Encapsulation hides the implementation details of the object; it exposes only the necessary functionality. Encapsulation frees the developer from the details of creating and managing the "scratch" array for each filter. Thus, encapsulation binds an implementation with its data. Polymorphism allows the developer to name an action for a hierarchy of classes but allows each class to implement it differently. Developers can act on objects of these classes as if they were all objects of the base class but invoke behavior specific to the actual class of the object. Polymorphism allows the definition of a filter to be completely separate from its execution. The operational code executes a filter without knowing its transfer function or digital algorithm.

[§] These instructions loop; the total number of steps is the product of the steps for one filter and the number of filters in the configuration.

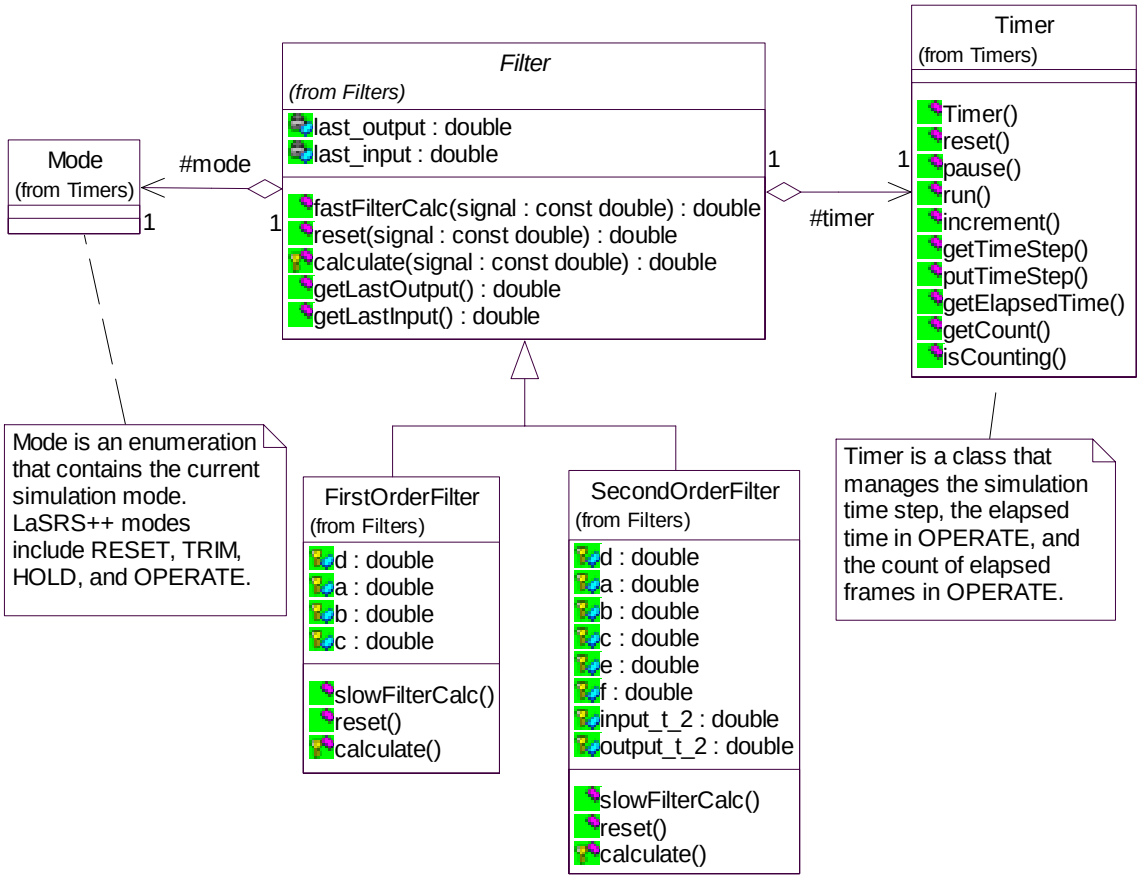

Figure 1 Filter Class Diagram

Figures 1 and 2 display the object-oriented filter design using unified modeling language (UML) notation [4]. Figure one shows a top-level view of the filter classes. Figure two illustrates how classes of digital algorithms (simple z-transforms in this case) are represented and associated with a filter. For ease of understanding, the figures show a simplification of the LaSRS++ design, focussing on filters using simple z-transforms as their digital algorithm[¶].

The design employs the strategy design pattern. The strategy pattern "defines a set of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from the clients that use it [6]." The Filter class is the base class of the hierarchy. As exemplified by the fastFilterCalc() method, the Filter class defines a filter to be an object that accepts an input, performs a calculation on the input, and produces an output. fastFilterCalc() consists of a case statement that performs reset() or calculate() based on simulation mode. reset() returns either the steady state output for the given input (default) or a user-defined initial output. calculate() executes the digital algorithm. reset() and calculate() are not defined; they are polymorphic methods. The derived classes must define them. The order of the transfer function distinguishes the derived classes. The order determines the number of Laplace-space coefficients that defines the filter (i.e. a, b, c, d, e, and f) and the amount of implementation data required. Thus, the order is the next level of commonality between filters.

Unlike the procedural functions, the filter classes are not subdivided by digital algorithm. The FirstOrderFilter and SecondOrderFilter classes can apply a variety of z-transforms because the z-transforms are also classes. In fact, the z-transforms are also modeled using

[¶] LaSRS++ also contains filters that use fixed-step integrators, include rate and position limiting, etc.

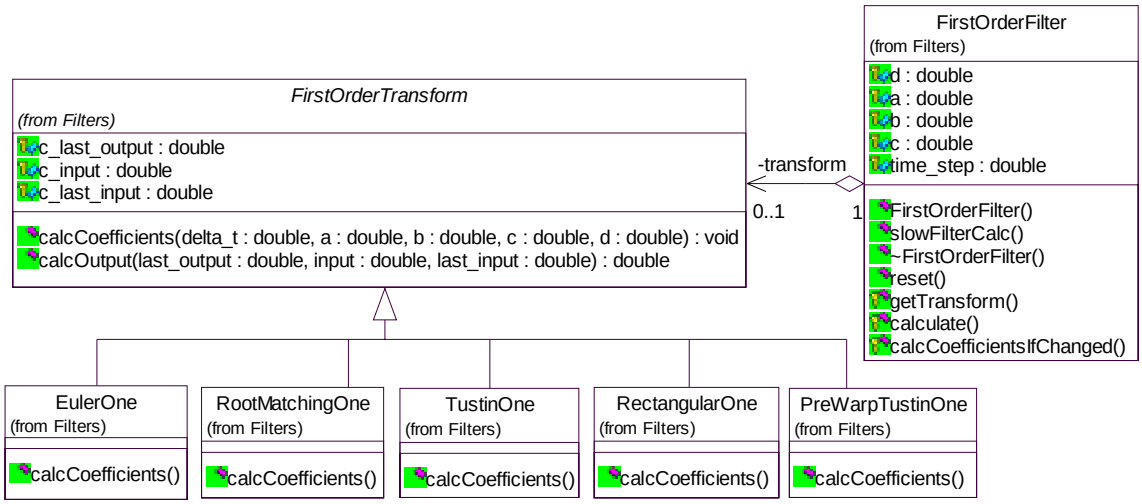

Figure 2 Z-Transform Class Diagram

the strategy design pattern. The filter object is paired with a z-transform when constructed; developers can mix and match filter objects with z-transforms[#]. These simple z-transforms are algorithms of the form:

Equation 1 Simple Z-Transform

$$y = ax + \sum_{n=1}^{O} \left(b_n \, x_{t - n\Delta t} + c_n \, y_{t - n\Delta t} \right)$$

where y is the output, x is the input, O is the order of the transfer function, and a, b, and c are coefficients. The coefficients are derived from the Laplace-space coefficients, the time step, and the z-transform. Since order determines the number of coefficients required, the z-transforms, like the filters, are categorized by order. Thus, a filter object can only be paired with a z-transform of the same order. Figure two shows the first order z-transforms[**]. The base class, FirstOrderTransform, encapsulates Equation 1 to first order. The base class holds three coefficients, one each for the input, last input (t-Δt), and last output (t-Δt). calcOutput() implements the equation. calcCoefficients() is a polymorphic method; its arguments demonstrate that the coefficients are purely a function of the Laplace-space coefficients and the time step. The sole purpose of the derived class is to define this coefficient function based on a given z-transform.

The Filter hierarchy stores the past values needed in Equation 1. The Filter class stores the last output and last input; all filters, regardless of order, require them. The SecondOrderFilter stores the additional past states that it needs (i.e. input and output at t-2Δt).

Encapsulation

Filter objects are defined when they are constructed. For example,

```
FirstOrderFilter filter_1(timer,
  mode, a, b, c, d, TUSTIN);
SecondOrderFilter filter_2(timer,
  mode, a, b, c, d, e, f, EULER);
```

The arguments are defined as follows: 'timer' is a reference to a class that contains the time step. 'mode' is a reference to an enumeration that represents the current simulation mode (i.e. RESET, HOLD, OPERATE, etc.). The letters a through f are the Laplace domain coefficients. The final argument is an enumeration that specifies the z-transform used by the filter. The argument list does not contain any "scratch" array. The

[#] LaSRS++ filters that utilize fixed-step integrators as their digital algorithm follow the same design. The integrators are encapsulated in classes that can be matched with a filter class.

[**] The design for the second order transform is identical except that the base class, SecondOrderTransform, contains five coefficients and its methods require two more double arguments.

"scratch" variables are coded into the filter and z-transform classes. A filter object contains the exact number of past values it requires. The z-transform contains the exact number of derived coefficients that it requires. To create the filter, the developer does not need to know much workspace a filter needs nor declare the workspace. The object-oriented design isolates the developer from that level of detail. The workspace variables are "encapsulated" in the class definitions. The object-oriented design eliminates the two potential defects associated with the scratch array in the procedural design: reading and writing past array bounds or more than one filter accessing the same scratch array. Also, the object-oriented design allocates the exact amount of workspace required for a given mix of filters. To avoid the array bounds defect, the procedural design encourages developers to allocate more workspace than necessary for a given mix of filters.

Polymorphism

In object oriented systems, a derived class can be assigned to and accessed through a reference[††] to its base class. Since FirstOrderFilter and SecondOrderFilter inherit from Filter, FirstOrderFilter and SecondOrderFilter objects can be assigned to Filter pointers. For example,

```
Filter* filter_1 =
      new FirstOrderFilter(timer,
      mode, a, b, c, d, TUSTIN);
Filter* filter_2 =
      new SecondOrderFilter(timer,
      mode, a, b, c, d, e, f, EULER);
```

Client code can only invoke methods in the Filter class interface through these pointers. However, polymorphism allows the derived class to redefine the behavior of methods in the base class interface. The reset() and calculate() methods in Filter are polymorphic. Executing filter_1->reset() actually calls FirstOrderFilter::reset(). Likewise, filter_2->reset() actually calls SecondOrderFilter::reset(). In the Filter design, the client does not call the polymorphic functions directly. Instead, the client calls fastFilterCalc(). fastFilterCalc() calls reset() or calculate() as appropriate for the current simulation mode[‡‡]. Thus, the control system code executes the filters as follows:

```
double output_1 =
  filter_1->fastFilterCalc(input_1);
double output_2 =
  filter_2->fastFilterCalc(input_2);
```

The syntax for both filters is identical even though the filters are defined with different transfer functions and different z-transforms. The first line will execute a first order filter using a Tustin z-transform. The second line will execute a second order filter using an Euler z-transform. No part of the filter definition appears in the two statements. Instead, filters are defined when constructed. The developer can change the order of the transfer function, the Laplace-space coefficients, or the digital algorithm without modifying the operational code. The execution of the filter is separate from its definition.

All filter definitions are collected into one block of code within the initialization unit. Viewing all of the definitions is simple. The developer does not have to travel from one filter execution to the next to observe all the definitions. The collected definitions can be viewed as a legend to the filters in the control system. The fact that filters must be named makes this notion more useful. Filter object names can correspond to names on a block diagram facilitating verification. In the procedural design, naming the output of the filter after the name on the block diagram provides a similar link; the block diagram name and the filter definition appear on the same line. However, the developer must still traverse hundreds of lines of code to verify all of the filters.

Being able to cluster the filter definitions also facilitates the creation and maintenance of multiple configurations. To define and manage multiple configurations, only one conditional construct is required in the initialization code. All the configurations and their associated filter definitions appear in the same block of code for easy identification. To add a new configuration, the developer simply extends the conditional construct and populates the new condition with filter construction

[††] A pointer or reference in C++.

[‡‡] In HOLD mode, fastFilterCalc() does not call either method; it returns the last output.

statements. No conditional statements need appear in the operational code; the complexity of the operational code is not increased by multiple filter configurations. The object-oriented design successfully allows the whole filter definition to be configured during initialization. The operational code is no longer forced to redundantly evaluate one-time initialization logic.

The following examples summarize the steps involved in modifying filters and creating a new configuration using the object-oriented design; they represent worst case conditions: order, coefficients, and digital algorithm are changed. The filter order is being increased in the example.

Changing an existing filter

1) Search the initialization code for the filter using its name.
2) Change the filter construction line to construct a filter according to the new definition.

Creating a new configuration

1) Search the initialization code for the conditional construct.
2) Expand the conditional construct for the new configuration.
3) For each filter, construct a filter object of the appropriate type and assign it to its corresponding Filter reference.

The steps for the object-oriented design are fewer than the steps for the procedural design. Changing an existing filter takes two steps in the object-oriented design versus seven in the procedural design. Creating a new configuration requires two one-time steps and one additional step for each filter in the object-oriented design. The procedural design can require eight steps for each filter. The possibility that a step will be neglected or performed incorrectly is reduced in the object-oriented design. Moreover, the steps only affect one code unit, initialization. Thus, there are fewer code units to review, test, and verify. Changes to fewer code units potentially leads to fewer re-compilations. Isolating the changes to the initialization unit also facilitates configuration management of the changes. Compared to the operational code, the initialization unit is usually smaller and changes less frequently because it focuses on one task, defining the variables in the control system. This simplifies the reconciling of changes from multiple developers. All these factors indicate that the object-oriented design is less costly to maintain than the procedural design.

Advantages of Z-Transforms as Objects

The discussion thus far concerns the maintenance of filters whose definition is immutable during operation. Many of the advantages that the object-oriented design holds over the procedural design disappear when the filter is mutable, i.e. the coefficients of the filter change during operation. In this situation, the order and coefficients of the filter must be known at the point of execution. The filter design reflects this restriction. Mutable filter calculations are handled by the slowFilterCalc() method. This method is not polymorphic. It is declared and defined in each derived class; it does not appear in the Filter class interface. The method can only be invoked by accessing it through the derived class. For example,

Initialization Code:

```
FirstOrderFilter filter_1(
  timer, mode, a, b, c, d, TUSTIN);
SecondOrderFilter filter_2(timer,
  mode, a, b, c, d, e, f, EULER);
```

Execution Code:

```
filter_1.slowFilterCalc(
  input, a, b, c, d);
filter_2.slowFilterCalc(
 input, a, b, c, d, e, f);
```

filter_1 and filter_2 are objects of the FirstOrderFilter and SecondOrderFilter classes; they are not references to the base Filter class as in the previous example. They implicitly identify the order of the transfer function wherever they are used. The coefficients appear as arguments to the slowFilterCalc() call; thus, they are also defined wherever slowFilterCalc() is execute.

However, the digital algorithm does not appear in the operational code as it does in the procedural design. The z-transform algorithm that the filter uses can be modified without changing the operational code. Configurations that differ only by the z-transform can still be defined in the initialization code. The procedural design still changes the operational code when the z-

transform is changed. Thus, the object-oriented design can still configure part of the mutable filter (i.e. the z-transform) in the initialization code; the procedural design forces all configuration logic into the operational code.

Polymorphism and encapsulation make this possible. How the z-transform and filter classes interact is a microcosm of how the filter and control system classes interact. The FirstOrderFilter accesses the derived z-transform class only through a reference to the FirstOrderTransform base class. The z-transform is defined only in the FirstOrderFilter constructor. The FirstOrderFilter constructs a z-transform according to the transform enumeration and assigns the z-transform object to a FirstOrderTransform reference. The filter's operational code only uses the reference; thus, the definition of the z-transform does not appear in the filter's operational code. The z-transform is configured once, in the filter's constructor call.

Performance

In a simulation environment, performance is an important factor in analyzing a design. Even though the object-oriented design facilitates maintenance, it must perform adequately to be a good design for simulation. The performance of the LaSRS++ filters was compared against those in LaSRS. The comparison was performed on an SGI Origin 2000. The SGI MIPSPro compiler version 7.2.1.3m was used to compile both the C++ and FORTRAN source. All files were compiled using the '–O2' optimization flag; no other optimization options were included. Performance was analyzed using the SGI Performance Analyzer. The performance analyzer was set to use program counter sampling and the R10000 hardware cycle counter. The test program ran each filter one million times using a predefined set of numbers for input. The performance tests were run ten times to verify consistency of the results. The median cycle count from the ten runs was used to calculate the average cycle count per call; the average was rounded to the nearest integer. The results appear in Table 1.

Table 1 C++ vs. FORTRAN Performance[§§]

Filter Type	Average Cycle Count	
	FORTRAN	C++
First Order Tustin	46	34
Second Order Tustin	50	43

The numbers demonstrate that the object-oriented design has performance comparable to the procedural design[¶¶]. Performance is not sacrificed by using the object-oriented design.

Conclusions

The object-oriented design facilitates maintenance of filters by moving the definition of a filter from its point of execution to the initialization code. In the initialization code, all filter definitions can be collected for easy inspection and modification. Multiple configurations can be created and managed by placing these filter collections into one conditional construct. This "one-time" configuration logic is also efficiently placed in the initialization code. The procedural design can force configuration logic into the operational code where it is redundantly evaluated each frame. The object-oriented design also shields the developer from the need to create and manage workspace data for the filter. Errors associated with active management of the workspace are eliminated. In comparison to the procedural design, the object-oriented design reduces the steps to change an existing filter or add a new configuration. Changes can be made rapidly, with few errors. The benefits of the object-oriented design are obtained without sacrificing performance.

[§§] For both the FORTRAN and C++ source, functions optimized for immutable filters were used, a.k.a. the "fast" filter calculations.

[¶¶] The performance tests were not exhaustive. Other compilers or other optimization flags could change the results. However, the results are sufficient to demonstrate comparable performance.

Bibliography

[1] Booch, Grady. *Object-Oriented Analysis and Design With Applications*. The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1994. ISBN 0-8053-5340-2.

[2] Stroustrop, Bjarne. *The C++ Programming Language*. Addison-Wesley Publishing Company, Reading, MA, 1997. ISBN 0-201-88954-4.

[3] *Working Paper for Draft Proposed International Standard for Information Systems – Programming Language C++*. ANSI X3J16/96-0225, 1996.

[4] Quatrani, Terry. *Visual Modeling With Rational Rose and UML*. Addison-Wesley. Reading, MA, 1998. ISBN 0-201-31016-3.

[5] Stevens, Brian L.; Lewis, Frank L. *Aircraft Control and Simulation.* Addison-Wesley Publishing Company, New York, NY, 1992. ISBN 0-471-61397-5.

[6] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John. *Design Patterns: Elements of Object-Oriented Software*. Addison Wesley Publishing Company, Reading, MA, 1995. ISBN 0-201-63361-2.

[7] Leslie, R.; Geyer, D.; Cunningham, K.; Madden, M.; Kenney, P.; Glaab, P. *LaSRS++: An Object-Oriented Framework for Real-Time Simulation of Aircraft.* AIAA-98-4529, Modeling and Simulation Technology Conference, Boston, MA, August 1998.